Structure-Preserving Reshape for Textured Architectural Scenes

Marcio Cabral, Sylvain Lefebvre, Carsten Dachsbacher, George Drettakis

EUROGRAPHICS 2009 / P. Dutré and M. Stamminger
(Guest Editors)

Volume 28 (2009), Number 2

Structure-Preserving Reshape for Textured Architectural Scenes

Marcio Cabral[†1] and Sylvain Lefebvre[‡1] and Carsten Dachsbacher[§2] and George Drettakis[¶1]

[1]REVES / INRIA Sophia-Antipolis
[2]Visualization Research Center / University of Stuttgart

Abstract

Modeling large architectural environments is a difficult task due to the intricate nature of these models and the complex dependencies between the structures represented. Moreover, textures are an essential part of architectural models. While the number of geometric primitives is usually relatively low (i.e., many walls are flat surfaces), textures actually contain many detailed architectural elements.
We present an approach for modeling architectural scenes by reshaping and combining existing textured models, where the manipulation of the geometry and texture are tightly coupled. For geometry, preserving angles such as floor orientation or vertical walls is of key importance. We thus allow the user to interactively modify lengths of edges, while constraining angles. Our texture reshaping solution introduces a measure of directional autosimilarity, to focus stretching in areas of stochastic content and to preserve details in such areas.
We show results on several challenging models, and show two applications: Building complex road structures from simple initial pieces and creating complex game-levels from an existing game based on pre-existing model pieces.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

The cost of developing modern interactive applications such as games is often dominated by the creation of a large number of detailed textured models. For many such scenes, such as typical "game levels", these models are often architectural, or more generally man-made, structures: In this paper we will be focusing our attention to this class of models. Traditionally, such assets are created by trained artists, who create the models and textures for each scene. While the tools used have improved in the past few years, this remains a tedious and painstaking manual process.

A different approach to create such content is procedural or grammar-based modeling [PL90, MZWG07]. While these methods hold great promise and can be very powerful, they require a "programmer-like" approach to modeling, making them hard to use, with a steep learning curve for typical modelers/artists.

Our solution allows the user to interactively modify or reshape *textured geometry*, since in the interactive applications we focus on a large part of the detail is usually incorporated in textures. Conceptually, our solution lies between the two methods discussed above. Both the geometry and the texture adapt to these modifications in an intuitive manner. This opens the way to easily creating large varieties of models from small sets of pre-existing model "pieces", as can be seen in Fig. 1. An additional motivation for our approach is that appropriate model pieces are becoming widely available as communities of modelers (or "modders") create and distribute them on a massive scale [EA08].

Our main design choice is to provide interactive feedback to the user while modifying textured geometry. Our approach is thus based on the definition of appropriate con-

† e-mail:Marcio.Cabral@sophia.inria.fr
‡ e-mail: Sylvain.Lefebvre@sophia.inria.fr
§ e-mail:dachsbacher@visus.uni-stuttgart.de
¶ e-mail:George.Drettakis@sophia.inria.fr

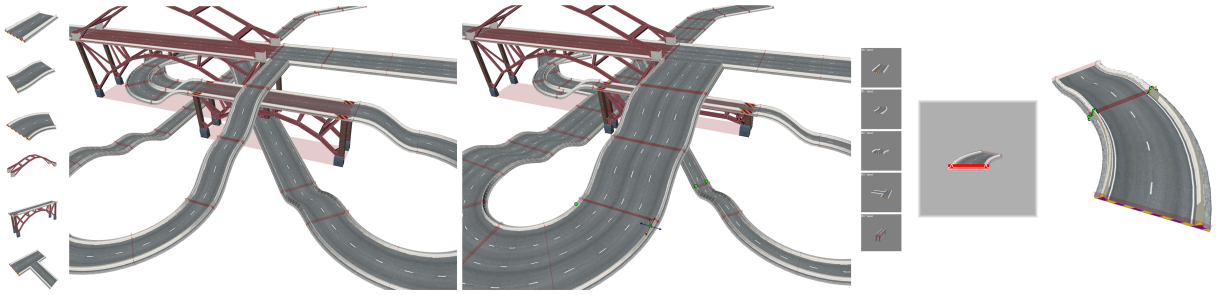

Figure 1: *Examples of our approach used to build road structures based on a small number of initial blocks. We show the building blocks (sides) and two example constructions made from these blocks in a few minutes, as shown in the accompanying video. Notice how pieces deform and adapt, and how texture replicates but also preserves detail in cases of stretching.*

straints and a fast least-squares solution. The interactivity of our approach gives sufficient flexibility to enable repeated tailoring of the constraints, thus allowing the user to obtain the desired result. In contrast to grammar-based approaches, which imply knowledge of model semantics, we do not assume any high-level knowledge of the model, e.g., that one part of the model is a door and the other a window. However, since we concentrate on architectural/man-made structures for interactive applications, the meshes do have an inherent "expected behavior" which we will seek to preserve. We thus base our reshaping operator on the key insight that angles are most important in keeping the aspect of a room or structure. Obvious examples are the horizontal orientation of the floor and the vertical orientation of walls. Given this choice, the main "degree of freedom" for the user will be the ability to make edges (of walls etc.) longer or shorter, without changing the angles. At the same time we restrict deformation for *small* edges, since they typically correspond to finer details, which we want to preserve.

We have designed and implemented a complete system to achieve these goals. Our main contributions are thus:

- A novel approach for deforming and reshaping architectural meshes, based on separating angular and length constraints. We propose a linear formulation of the problem, solved as a least square minimization. Our approach allows interactive geometry reshape with intuitive results.
- A reshaping tool tightly coupling geometry and texture. While the user manipulates the geometry, texture features are updated so as to maintain their visual appearance while following the deformations; to our knowledge such coupling has not been done before.
- The use of directional autosimilarity to identify regions of a texture which can be appropriately deformed, while keeping structured parts rigid during interactive textured geometry reshape.
- An interactive method to re-introduce texture detail in stretched regions, based on detail extraction, tiling and a realtime rendering solution.

We have implemented the above ideas in an interactive system. As we will show in our results and applications (Sec. 5), our new method provides an interactive approach to complex reshaping of textured models (see Fig. 1).

2. Previous work

Our work involves ideas from several areas: procedural modeling of geometry, example-based geometry modeling, mesh editing and texture synthesis. We describe next the key ideas most related to our work.

Procedural modeling Several approaches have been proposed for automatic geometry synthesis. L-Systems build geometry from a rule set. They have been applied to plants [PL90], cities [PM01], buildings [MWH*06] and facades [MZWG07]. Similarly, geometric languages let the user write programs generating complex shapes from simple operations [BFH05]. While such approaches often provide very impressive results, the main drawback is the level of expertise they require: The rules have to describe how every single geometric primitive is to be placed in the scene. Despite impressive results with visual interfaces [LWW08], understanding of rules and grammars is still necessary to create/modify initial models to a certain extent.

Modeling by example Merrell [Mer07] proposed an example-based scene synthesis method producing impressive results. The example is given as a set of building blocks aligned on a regular 3D grid, which evidently have to be very carefully modeled. By reproducing the neighboring relationships of the input blocks in the output, the algorithm automatically generates larger randomized environments. The method was recently adapted to handle arbitrary inputs [MM08].

Other approaches have been proposed to model from examples, in particular by assembling mesh pieces. Funkhouser et al. [FKS*04] let the user slice parts from objects and assemble them in new ways. Kraevoy et al. [KJS07] follow a similar approach. Both methods are

targeted at creating objects and would be difficult to adapt for entire environments. A similar approach is used by Zhou et al. [ZHW*06] at a much finer scale to synthesize mesoscale structures. Patches of geometrical details are carefully stitched together to cover a surface.

Mesh editing Mesh editing techniques such as Poisson mesh editing [YZX*04] and Laplacian surface editing [SCOL*04] provide interactive tools to deform a mesh while retaining its overall appearance. These approaches work very well as long as the mesh is smooth and finely tessellated. This does not hold for architectural pieces, where the tessellation is often irregular and sharp edges are common. Our approach is inspired by these works and it provides a new formulation better suited to our needs.

Constrained editing In earlier work, Gleicher [Gle92] outlined the benefit of mixing direct manipulation with constraint solvers. Spatial relationships between objects are inferred from user manipulations and later maintained by the system. Similarly, Xu et al. [XSF02] helped the layout of many objects in a scene by guiding user manipulation using constraints. We follow a similar trend: We let the user manipulate a scene while the system automatically updates vertex positions and textures through constraints.

Texture and image resizing There has been much recent work on texture and image resizing. The closest approach has been proposed by [WTSL08]. A grid is used to deform an image while preserving gradients. The method could be adapted to our needs, although our emphasis is more on structure vs. detail rather than salience. Seam carving [AS07] minimizes energy to find the appropriate image seams to remove, effecting image resizing. Tai et al. [TBTS08] use texture synthesis from example [WL00] to recover details in stretched image areas. This is related to our detail sliding idea Sec. 4.5, with the key difference that our approach must allow for interactive feedback. A method for model resizing has been recently proposed [KSCOS08]. However, memory and computation time implied by the 3D grid used there would be a big handicap for our application.

In contrast to the above approaches, we consider geometry and texture reshape together. In Sec. 3 we explain how to define the constraints to maintain edge directions and other desirable properties and how we efficiently solve for *geometry reshape*. In Sec. 4, we introduce *directional autosimilarity* and show how it is used to *reshape texture*. Our approach preserves structured parts of the texture while re-introducing detail in stretched regions. We present examples and applications in Sec. 5 and discuss limitations in Sec. 6.

3. Geometry reshape

Our goal is to reshape an architectural model while retaining its characteristic features. As explained above, we consider that preserving angles is the most important constraint to impose for the class of models under consideration.

In our formulation, vertices are either *variables*, in which case the solution to our system determines their position, or they are *constrained*. Constrained vertices can be manipulated by the user - in which case their position is attached to the mouse ("handles") - or they remain fixed. In Fig. 2 and the video constrained vertices are shown in green.

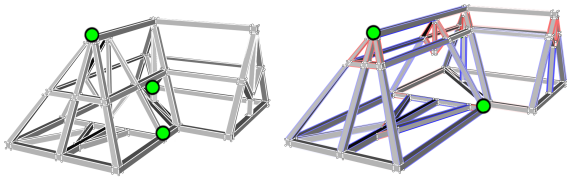

Figure 2: Left: *The original model.* Right: *The final result. Elongated edges are highlighted in blue and compressed edges in red. This motion is achieved by moving the handle up (green point in the center).*

Our approach has three major steps:

- The input model is loaded and organised into a simple constraint graph (Sec. 3.1).
- A set of constraints is defined capturing the relationships between vertices, walls and edges (Sec. 3.2). In addition to preserving angles, we maintain short edge length, vertex/face contacts and avoid edge flips.
- A solver recomputes vertex positions from the user controlled handles, while attempting to preserve the constraints (Sec. 3.3).

The two first steps are a pre-process. In contrast, the solver is used at run-time, during user manipulation of the model. To enable interactivity, we rely on a simple solver and allow it to fail or refuse user input if this leads to degeneracies (ie., collapsing edges or collisions). We argue this behavior is reasonable since the user has full freedom to assign new handles and guide the solver in avoiding degeneracies, thus obtaining the desired result. Nevertheless, we propose simple mechanisms to help the user in this task (see Sec. 3.4).

3.1. Input and graph construction

We first load the model and create a corresponding *constraint graph*. We assume that the input is a textured mesh, in the form of an indexed face set. We expect triangles to be grouped in textured surfaces, which we refer to as *surfaces*. Triangles within a same surface share vertices; a shared edge implies that vertices share 3D and UV coordinates. The nodes of the constraint graph are the vertices of the model. Note however that two co-located vertices in different textured surfaces will share a same graph node.

We distinguish three types of edge within a textured surface. *Contour edges* are used by a single triangle in the textured surfaces, while *angle edges* are shared by two non coplanar triangles. Both edge types are added to the graph.

Lastly, *flat edges*, shared by two co-planar triangles, are ignored. In addition, textured surface contours must be well formed: i.e., a contour is formed by following the ring of adjacent contour edges. We also support holes.

Finally, we identify connected components of the graph. In the examples we show here, the number of connected components is typically low (most often around 3 or 4).

3.2. Constraints

We use the constraint graph to express the properties to be preserved whenever handles are being moved. We thus formulate a system of equations expressing constraints that must be either strictly enforced or minimized. In what follows, we will present "strict" and "soft" constraints. Table 2 and table 3 provide a summary. We will explain how these are actually used in Sec. 3.3.

Notations In what follows v_i is a vertex and E_{ki} an edge between vertices v_i and v_k. N_i is the set of neighboring vertices of vertex i and $|N_i|$ the size of this set. We designate variables using a "tilde" symbol. Hence, $\tilde{v}_i$ is the unknown next position of vertex i, while v_i simply refers to its initial position. We note $u_{ki} = \frac{v_i - v_k}{||v_i - v_k||}$ the normalized direction of edge E_{ki}, while $l_{ki} = ||v_i - v_k||$ is its length. Please see Figure 3 for more details.

Note that the edge direction u_{ki} and length l_{ki} are fixed with respect to variables since they are computed on the initial graph; they are thus constants in the system being solved (Sec. 3.3). Finally $v_{T_i^j}$ is the i-th vertex of triangle j, and n_{Tj} the normal to triangle j. Please refer to Table 1 for a summarized list of notations.

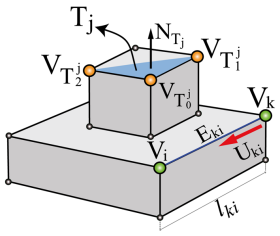

Figure 3: *Notations used in the paper.*

Edge direction constraints The most important constraint is to preserve angles between planar faces. Rather than working directly on faces, we equivalently preserve edge directions. For each vertex i we derive the following equation:

$$|N_i|\tilde{v}_i - \sum_{k=0}^{|N_i|-1} (\tilde{v}_k + (u_{ki} \cdot (\tilde{v}_i - \tilde{v}_k))u_{ki}) = 0 \qquad (1)$$

Intuitively, this simply states that from any neighboring vertex k we can come back to vertex i by adding the appropriate

Notations	
v_i	initial position of vertex
E_{ki}	edge between vertices v_i and v_k
N_i	set of neighboring vertices of vertex i
$\|N_i\|$	size of the set N_i
$\tilde{v}_i$	vertices as variables
$u_{ki} = \frac{v_i - v_k}{\|v_i - v_k\|}$	normalized direction of edge E_{ki}
$l_{ki} = \|v_i - v_k\|$	length of normalized edge u_{ki}
$v_{T_i^j}$	i-th vertex of triangle j
n_{Tj}	normal to triangle j

Table 1: *Notations used in the paper*

length in the direction of E_{ki}. Note that u_{ki} is constant and computed on the initial mesh, while $\tilde{v}_i$ and $\tilde{v}_k$ are variables.

An important property of this equation is that it does not restrict the edge length but only the alignment of the vertices. However, vertices are free to move so the solver might have to compromise and change the direction of some edges (see Figure 5(right)).

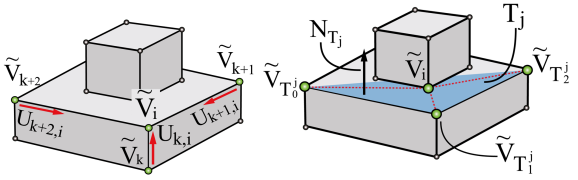

Figure 4: *Edge direction constraints (left) and contact constraints (right).*

Edge length preservation It is also desirable for the edges to keep their original lengths: The result of our geometry reshape will thus be as close as possible to the initial mesh.

We express this with an *edge stress* term S. We thus seek to minimize the following term for each edge:

$$S_{ki} = w_{ki} \ (u_{ki} \cdot (\tilde{v}_i - \tilde{v}_k) - l_{ki}) \qquad (2)$$

The scalar term w_{ki} controls how well the edge length must be preserved, or edge *stiffness*. We typically give a higher importance to small edges compared to large edges, and consider edges shorter than a user defined threshold as rigid.

We compute w_{ki} as:

$$w_{ki} = w_{small} + (w_{long} - w_{small}) \left(\frac{l_{ki} - l_{min}}{l_{max} - l_{min}} \right) \qquad (3)$$

where l_{max} is the largest edge length, l_{min} the threshold below which edges are considered rigid, and w_{small}, w_{long} control the overall edge stiffness. We use $w_{small} = 10^{-3}$ and $w_{long} = 10^{-5}$. Note that w_{small} is larger than w_{long} to make small edges more rigid. For rigid edges, we set $w_{ki} = 1$

We illustrate direction constraints and edge length preservation in Fig. 2. Our approach preserves angles and the

length of short edges, while allowing large edges to be either shortened or lengthened.

Additionally, all edge lengths must remain positive during reshaping, which is expressed with the following strict constraint per-edge:

$$u_{ki} \cdot (\tilde{v_i} - \tilde{v_k}) > 0 \tag{4}$$

Contacts Often structures lie on other surfaces: Pillars, doors, windows, etc. Using only edge direction constraints, we would not be able to capture these relationships.

Within a connected component, coplanarity is already captured by edge direction constraints. However, two disconnected components do not share triangles: Contact relationships are not captured by the previous equations. We check whether vertices of one component are on a triangle of another component. When this happens we add one "strict" constraint for co-planarity, simply using the vertex and the first vertex of the triangle containing it:

$$(\tilde{v_i} - \tilde{v}_{T_0^j}) \cdot n_{T^j} = 0 \tag{5}$$

where n_{T^j} is the normal to triangle T^j, $\tilde{v_i}$ and $\tilde{v}_{T_0^j}$ belong to different connected components. We also add three "soft" constraints to encourage the vertex to stay at the same distance from the triangle vertices:

$$\left((\tilde{v_i} - \tilde{v}_{T_k^j}) - (v_i - v_{T_k^j})\right) = 0,\ k = 0..2 \tag{6}$$

These are mandatory otherwise the system is under-constrained, since co-planarity does not tells us "where" the vertex should be on the plane. Please see Figure 4 for more details.

Edge groups Architectural models contain many implicit constraints. For instance, the height of doors is typically the same throughout a building. In general, inferring this type of constraint from input geometry is a difficult problem and depends on the semantics of the model.

We do not infer semantic information; instead we use some easily identifiable properties of the model, notably groups of *similar* edges. This provides correct default behavior for most cases, and it may be switched off by the user at any time. Specifically, we define an *edge group* as a set of edges having similar direction, similar length, and being spatially close to each others. This is achieved using a simple clustering approach.

For each edge group we add constraints stating that edges must keep similar length. The first edge of the group is used as reference. For each other edge we write the equation stating that its length must be equal to the length of the first edge, using a *group error* G_E:

$$G_E = w_g \left(\left(\sum_{E_{ji} \in G} u_{ji} \cdot (\tilde{v_i} - \tilde{v_j}) \right) - |G|\ u_{10} \cdot (\tilde{v_0} - \tilde{v_1}) \right) \tag{7}$$

We use v_1, v_0 to denote the vertices of the first edge of group G, and u_{10} for its direction. The edge between v_j, v_i (E_{ji}) is within the same group. The group has $|G|$ edges. We use $w_g = 0.1$. An example of the effect of edge groups is illustrated Fig. 5.

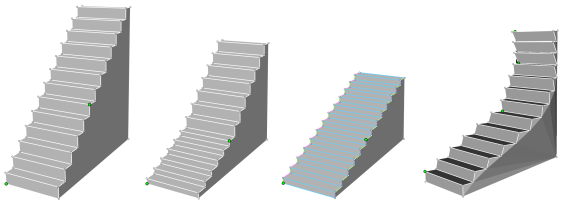

Figure 5: Left: *A model of stairs.* Middle-left: *Result obtained if we do not use edge groups and use the handles shown in green. Unwanted distortion appears.* Middle-right: *Using our edge group approach, we achieve uniform reshape of the stairs.* Right: *The solver changed edge directions in order to achieve the motion required by the user.*

Edge direction	$\lvert N_i \rvert \tilde{v_i} - \sum_{k=0}^{\lvert N_i \rvert - 1} (\tilde{v_k} + (u_{ki} \cdot (\tilde{v_i} - \tilde{v_k}))u_{ki}) = 0$
Edge > 0	$u_{ki} \cdot (\tilde{v_i} - \tilde{v_k}) > 0$
Co-planarity	$(\tilde{v_i} - \tilde{v}_{T_0^j}) \cdot n_{T^j} = 0$

Table 2: *Summary of desired strict constraints.*

Edge length	$w_{ki}\ (u_{ki} \cdot (\tilde{v_i} - \tilde{v_k}) - l_{ki})$
Contact positions	$\left((\tilde{v_i} - \tilde{v}_{T_k^j}) - (v_i - v_{T_k^j})\right),\ k = 0..2$
Edge groups	$w_g \left(\left(\sum_{E_{ji} \in G} u_{ji} \cdot (\tilde{v_i} - \tilde{v_j}) \right) - \lvert G \rvert\ u_{10} \cdot (\tilde{v_0} - \tilde{v_1}) \right)$

Table 3: *Summary of soft constraints to be minimized.*

3.3. Solver

As discussed above, we have three types of constraints: inequality, equalities and terms to that should be minimized. For the first, an accurate solution would require linear programming, thus sacrificing interactivity. We discuss our alternative solution in Sec. 3.4. For the remaining constraints a typical way to solve is to describe the problem as an least squares minimization and solve with efficient linear system solvers enabling interactivity. We can write this more formally as:

$$x' = argmin_x ||Ax - b||^2 \tag{8}$$

where x is the vector of positions $\tilde{v_i}$ of all the vertices in the model except for the handle(s), whose position is given by

user input and the fixed vertices which are not affected by the solution. A is the matrix defined by the constraint equations.

A typical way to enforce equality constraints in a least square optimization is to include them in the system with a large weighting term [Loa85]. This can lead to small inaccuracies, which are tolerable in our context.

In this system, we weight the minimization constraints with appropriate values w_i. In particular we use weight $w_g = 10^{-2}$ for Eq. 7, $w_c = 10^{-5}$ for Eq. 6; for Eq. 2 weighting is embedded in w_{ki}. In practice, these weights work across most of our meshes. Only l_{min} – fixing the edge length under which $w_{ki} = 1$ (Sec. 3.2) – needs to be adapted since it strongly impacts the reshaping behavior.

We compute the normal equations, pre-factor the sparse matrix $A^T A$ using the Cholesky factorization of the TAUCS library [TCR03], and solve very efficiently at run-time. Such approaches have been used for interactive mesh manipulation [BBK05, BS08].

3.4. Edge flips and User Constraints

To deal with the non negative edge inequality (Eq. 4), we exploit the interactive aspect of the user manipulations: As the user drags handles only small motions occur. After every step we verify that no edge is collapsing. If an edge becomes too small we artificially increase the length l_{ki} (Eq. 2), making it less likely to collapse. Since this only affects the constants in the system, it does not affect the interactive performance of our approach. If an edge does collapse we refuse the last motion and rollback to the previous position.

Note that linear programming *could* be used to enforce this inequality, for instance by incrementally updating constraints during interactive manipulation [BMSX97]. In practice interactive feedback makes it easy to detect and avoid degeneracies during manipulation: It seemed unnecessary to resort on more complex solvers in our context.

User-defined additional constraints In some cases (e.g., Fig. 6(left)), it may be necessary to add additional constraints. Our system provides a simple mechanism to link two vertices, resulting in an additional constraint. An example is shown in Fig. 6(right).

3.5. Limitations

One of the main limitations is that we ignore surface interpenetration, which can result in internal structures going through walls.

The behavior of the reshape depends on the chosen handles, and an unfortunate choice may result in undesired motions. Thankfully, given the interactive nature of our approach the user can quickly correct for such cases, as illustrated Fig. 11. Note that the system behaves reasonably even if the user ask for deformations where edge direction cannot be preserved, as illustrated Fig.5(right).

Many of these limitations could be addressed by adding constraints; however, we cannot infer all these constraints since they often depend on model semantics.

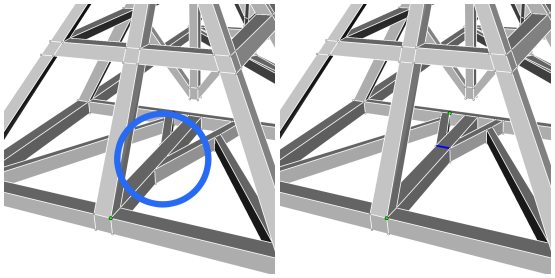

Figure 6: Left: *parts of the roof frame (in the blue circle) should move together; however they do not share edges or vertices.* Right: *The user can add an a link between appropriate vertices, resulting in appropriate deformation.*

4. Texture reshape

The vast majority of interactive applications enhance the appearance of objects with texture mapping. But surprisingly most existing geometry editing approaches do not provide special treatment of textures during deformation. For single-material models texture synthesis from example methods could be used to automatically obtain a new texture map [WL00]. However, on architectural models textures are often much richer, contain many architectural elements: A door frame, various decorative elements and other wall details. It is therefore very important to retain and preserve the appearance of this information when the user changes the geometry of a mesh, even though the only information we have are the texture pixels.

We thus propose a new approach to resize texture maps targeted at architectural environments, inspired by ideas from image warping [WTSL08], image completion [DCOY03] and texture synthesis from example [EL99]. It is designed to perform efficiently during interactive manipulation of the geometry, while providing high quality results. Our work is most similar to the approach of [WTSL08] in that it deforms the texture to concentrate stretch in some particular areas. However we cast a different formulation based on the new notion of *directional autosimilarity* (Sec. 4.2).

In the following we use *texture map* to designate the image mapped onto a surface and *stochastic texture* to designate areas of the image having homogeneous content.

4.1. Overview

Consider the texture in Fig. 7(a); if we enlarge the geometry with no special treatment, we get the result of Fig. 7 (b). We want to get the result shown in (c), where structured parts of the texture are preserved, and stretching is focused on the stochastic texture regions.

To do this we use *directional autosimilarity*, which measures whether a texture region remains similar to itself if slightly translated along a given direction. We typically compute this measure along the two main image axes. Our autosimilarity measure often takes large values *across* edges and low values in the direction *parallel* to edges (see Fig. 7(d)). In areas of stochastic content it exhibits a low value - compared to edges - in all directions. This provides an effective measure to detect regions of similar homogeneous content.

We encode this measure in an autosimilarity map (Fig. 7(d)). We then warp the texture using a grid, shown in Fig. 7(e). The goal is to first replicate edge-length changes from geometry reshape to texture space, and then to deform the grid guided by the autosimilarity measure, giving the result shown in Fig. 7(f). We use the result to generate a map of distorted texture coordinates for the stretched polygon.

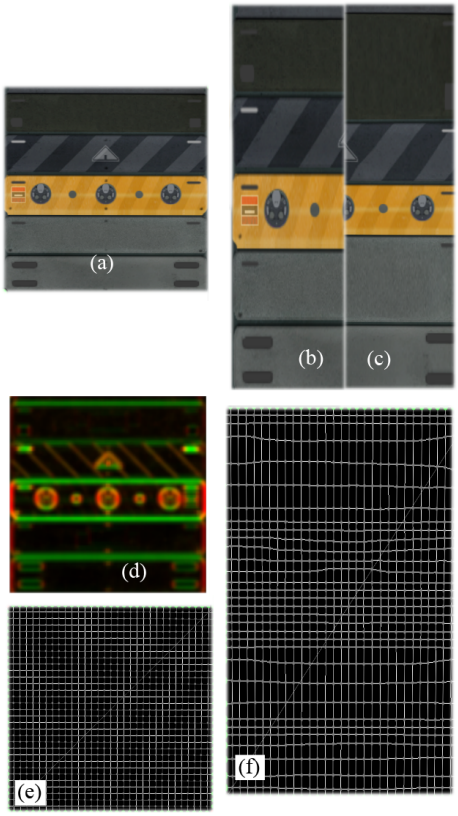

Figure 7: *(a) The original textured polygon. (b) The polygon has been stretched without reshape. (c) The desired result, where features are preserved. (d) Directional autosimilarity map. (e) The grid on the original texture. (f) The deformed grid. Note how rigid features are preserved.*

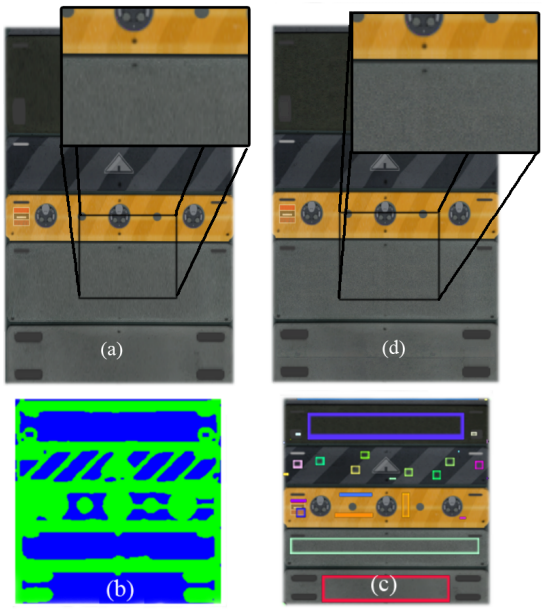

Figure 8: *(a) Stretched texture: inset shows blurred detail. (b) Texture segmentation into rigid/stochastic regions. (c) Detail tiles extracted from original texture (color rectangles) (d) Stretched texture: inset shows re-introduced detail, extracted from tiles. (Please zoom to see details)*

As we can see in Fig. 8(a), details in stretched regions are now blurred. We want to reintroduce detail by finding and using appropriate regions in the original texture. We do this by segmenting the texture into rigid/stochastic regions (Fig. 8(b) and then extracting *detail tiles* Fig. 8(c). These are regions of stochastic content which can be tiled to reintroduce detail into stretched regions. We introduce an appropriate rendering technique that can use this information, giving the final result shown in Fig. 8(d).

4.2. Directional texture autosimilarity

Architectural texture maps often contain a mix of structural elements and areas of homogeneous, stochastic texture content. That is, for a given pixel neighborhood in these areas other, similar pixel neighborhoods can be found in its vicinity. Since we know these textured areas are easier for us to reproduce, our goal will be to detect them so as to focus stretch onto them.

When the geometry is deforming, the edges of the textured polygon will change length (Sec. 3.2). Often these changes will be anisotropic, for instance when a wall is

widened while keeping its height. The structural elements contained in textures are often parallel or orthogonal to the ground plane - a plinth, a door frame or brick walls are good examples. These structural elements are likely to be very auto-similar in a given direction along which they can be considered as a texture. This implies that stretch is not likely to be visible if applied in this same direction - however it must be prevented in other directions.

In order to exploit these degrees of freedom, we introduce the notion of *directional autosimilarity*. We start from a given set of directions, typically the vertical and horizontal texture space axes. For each direction we compute an error map indicating whether each pixel location can be considered as an autosimilar texture *in the given direction*. This is done by comparing small neighborhoods along a 1D line centered on the pixel and oriented parallel to the current direction.

More formally, we define an error at a given pixel p expressing how far we are from autosimilarity:

$$T_{\vec{dir}}(p) = \frac{1}{2\Delta} \sum_{\substack{q=p-\Delta\vec{dir} \\ q\neq p}}^{p+\Delta\vec{dir}} ||N(q) - N(p)||^2 \qquad (9)$$

where $\vec{dir}$ is the considered direction, Δ controls how far around the pixel we search and $N(p)$ is a pixel neighborhood around p. We typically use 5×5 neighborhoods and a value of $\Delta = 10$. The result for two directions can be seen in Fig. 7(d), encoded as red and green.

4.3. Deformation grid and Constraints

We now exploit directional autosimilarity maps to deform the texture map. We achieve this using a *deformation grid* overlaid on the texture map. We intersect the grid with the contour of the surface polygon in texture space, using a DDA approach for robustness. This is illustrated Fig. 7. The spacing of the grid is chosen to be as large as possible while still preserving the features of the texture map. We typically use a spacing of $\frac{1}{32}$ in normalized texture space.

We call *nodes* the vertices of the grid. We distinguish three types of nodes: corner nodes, denoted c_i, boundary nodes, which lie on the edge of the polygon boundary but are not corners, denoted b_i, and interior nodes (Fig. 7(d)). We use g_i to denote any kind of grid node (interior, boundary or corner).

We deform this grid in two steps, first solving for the shape of the textured polygon (i.e., the corner nodes) and then for the boundary and interior nodes.

Grid corner constraints First, in order to determine the new shape of the polygon in texture space we replicate the changes in length of the world space polygon edges. We deform the texture space polygon using a gradient-based approach [BS08]. Second, as we shall see, for efficiency we impose that boundary edge directions are maintained. Finally, we fix the position of one corner node to remove the translational degree of freedom of the system.

We use the same convention as for geometry reshape where $\tilde{c_i}$ denotes the (variable) position of corner i during reshape. For gradient-based deformation of the triangles we consider each corner c_i, $i = 0..2$ of each triangle T in the textured surface. We define the set of $N_T(c_i)$ of neighboring corners c_k, with $|N_T(c_i)|$ the size of this set. We thus have:

$$|N_T(c_i)|\tilde{c_i} - \sum_{c_k \in N_T(c_i)} [\tilde{c_k} + r_{ki}\,(c_i - c_k)] = 0 \qquad (10)$$

where r_{ki} is the length change ratio of edge ki from the geometry reshape.

The interior node system, defined in the following section, depends on the direction of the polygon contour edges. To avoid having to refactor the interior system during interaction, we constrain the edges of the contour to maintain their direction, using a 2D version of Eq. (1), replacing v_i by c_i.

Interior grid node constraints The first constraint concerns boundary nodes b_k, contained in the edge defined by corners c_i, c_j. We define $N_{c_ic_j}$ to be the normal direction to the edge defined by c_i, c_j. We require that the b_k remain on the edge, by imposing the following constraint:

$$\tilde{b_k} \cdot N_{c_jc_i} = \tilde{c_i} \cdot N_{c_jc_i} \qquad (11)$$

Note that thanks to the edge direction constraint described above, normal $N_{c_jc_i}$ will never change during interactive manipulation.

For interior nodes we want to achieve a deformation which will preserve rigid regions and concentrate stretching in stochastic regions. For each edge defined by grid nodes g_i, g_j and for each deformation direction $\vec{dir}$ (there are typically two), we define energy e_{ij} to be minimized:

$$e_{ij} = \left((\tilde{g_i} - \tilde{g_j}) \cdot \vec{dir} - (g_i - g_j) \cdot \vec{dir}\right) s_{ij} \qquad (12)$$

We define the stiffness weight s_{ij} using the autosimiliarity error map $T_{\vec{dir}}$ in direction $\vec{dir}$ as:

$$s_{ij} = s_{min} + T_{\vec{dir}}\left(\frac{v_i + v_j}{2}\right)\left(1 + s_{align}\left(1 - |(v_j - v_i) \cdot \vec{dir}|\right)\right) \qquad (13)$$

where T is accessed with a 2D coordinate in texture map space, s_{min} is the stiffness of texture areas, s_{align} controls how much orthogonal alignment must be preserved. Note that we equalize luminance in all textures, and we assume that lighting information is not included.

4.4. Online Reshape Solver

To achieve online texture reshape we use an approach analogous to that used for geometry (Sec.3.3). We express all constraints as a linear system and weight equations with respect to their importance. We use a direct least square solver

(sparse Cholesky factorization [TCR03]). For fast interactive manipulation we pre-factor the matrices for both the contour and interior system. During interactive manipulation, only the vector b needs to be recomputed and $A^T b$ updated, where A contains the constraint equations.

An example is shown in Fig. 7 where the textured surface has been stretched significantly. For efficient storage and display, we manipulate and store uv coordinates rather than the texture itself. The deformation is thus coded in these coordinates, which are rendered into a render target and stored as a texture of the same resolution as the original texture map. We call this the *distortion map*. During rendering we thus use one level of indirection to access the original texture.

Regions where stretching has occurred will of course be blurred, since the texture hardware interpolation will be used. This is illustrated in Fig. 7(b). To alleviate this problem, we present a new approach to reintroduce details for the reshaped texture.

For toroidal textures we want to have an integer number of tiles covering the polygon. We solve for the corners, then extract a bounding square in texture space. This allows us to compute the number of repetitions in UV and the amount of distortion for a single tile (not to be confused with the detail tiles described below). We then solve as usual for the interior nodes. Windows on the right of Fig. 9 are repeated in this manner.

4.5. Reintroducing detail

The first step in recovering detail is to identify within the texture which regions can be used as a model to reintroduce details. We first perform a binary segmentation of the texture into stochastic and structural areas following our autosimilarity measure. We then identify stochastic regions and "grow" a rectangular tile within each such region. These tiles will be used to reintroduce detail during rendering. Figure 8 illustrates this process.

Texture segmentation and Detail extraction To segment, we first merge the directional autosimilarity error maps (typically we have two, one along each axis), keeping the largest error for each pixel. This gives us a new map measuring how rigid each pixel is in the worst case direction. We apply a binary segmentation [BJ01] onto this map. After experimentally setting the parameters of the segmentation we use the same ones on all textures of our dataset. This works well in practice since our textures all have equalized luminance and do not contain lighting information.

The segmentation separates the texture in disconnected regions (see Fig. 8(b)). We assume that each region corresponds to a stochastic texture with locality and stationary properties [WL00] - note that while this is not to be expected in general, it works well in our cases since most architectural textures depict regions of homogeneous materials separated by structural elements.

Within each region we seek to extract a rectangular tile representative of the details, with the goals of using this to re-introduce detail when stretching occurs. Starting from a random seed we grow the rectangular tile in a "spiral-like" manner. We perform several iterations and keep the largest rectangle as our tile (Fig. 8(c)). This naive search could be improved using more involved algorithms [DMR97].

For each texture, we finally create a *tilemap* which indicates in each pixel the tile ID associated with its region. The tile ID indexes a small table which contains the upper and lower corner of each tile. If a pixel belongs to a rigid region, the tile ID is set to a special value (for example 0).

Online Detail sliding After solving the system for the polygon, we obtain new texture coordinates for each vertex. During rendering of the final scene, these per-vertex texture coordinates are interpolated by the rasterizer and passed over to the fragment shader. We use these coordinates to access the deformation map (see Sec. 4.4), retrieving the coordinates computed by the deformation grid. Directly accessing the original texture with these coordinates produces blurred details.

Instead, we perform a lookup in the tilemap to determine whether the pixel has an associated tile. If yes, we perform a lookup in the texture from the tile, using a modulo operation to let the tile cycle in heavily stretched regions.

Directly using colors from the tiles would produce artifacts at the boundary of stretched regions. Instead, we separate the original image into a base and a detail layer using a bilateral filter [TM98]. We only use the tiles on the detail layer, the base layer being stretched as previously. However in the base layer stretched areas contain no detail.

Finally, to avoid modifying the original texture when no stretch is applied we gracefully transition from the original detail layer to the tiled details using a measure of the local stretched. This is easily computed from the deformation grid.

The shader pseudo-code below summarizes these operations:

```
UV = distortionMap(p); // p contains the distorted coordinates
tile = tilemap(UV); // UV accesses tilemap & original texture
if( tile != 0 ) { // pixel isnt rigid
    detailTile = lookup(tileMap, p modulo tileSize);
    color = base(UV) + lerp(detail(UV), detailTile, stretch);
}
else
    color = base(UV) + detail(UV);
```

The result of this operation is that detail is added in the stretched regions. In addition, the modulo operation during the lookup of the tilemap will result in tiles being repeated when necessary. We illustrate these operations in Fig. 8.

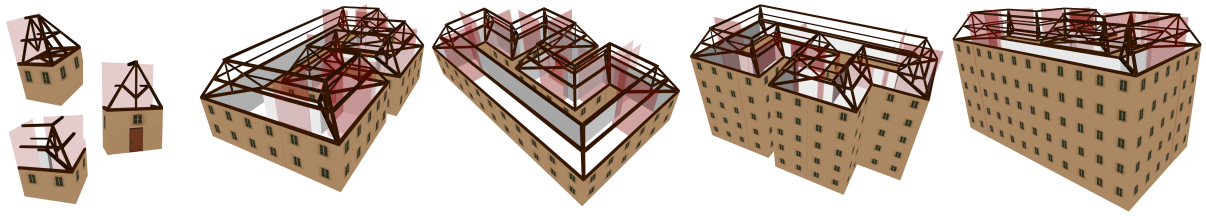

Figure 9: *An example of houses with a complex roof frame using three pieces shown left. We show two views of two different variants of the resulting house.*

Piece	vars	nnz A	nnz A^tA	nodes	edges	eqns	factor	solve
	114	2136	4284	38	60	217	4 ms	1 ms
	1314	30294	60936	438	760	3874	2155 ms	10 ms
	258	5604	13068	86	176	517	44 ms	1 ms
	909	15588	30681	303	530	1555	175 ms	3 ms

Table 4: *Performance measure for some of the meshes utilized in the paper.* nnz *stands for 'number of non-zero entries',* vars *for variables and* eqns *for equations.*

5. Results and Applications

The geometry (Sec. 3) and texture (Sec. 4) reshaping algorithms constitute the core of our approach. We next present present three examples using a "mesh piece modeling" application. To truly appreciate our results, please see the accompanying video.

Texture and geometry reshape can be used together to form a powerful tool for the edition and creation of architectural scenes. The user can create complex models based on a small set of initial pieces, which are either specifically built for this purpose, or are easily available (e.g., the pieces of a game level in our examples).

We assume that each piece has "portals" associated at each extremity, and that portals are compatible, i.e., have the same number of vertices. This is for example the case with the game level pieces we extracted.

The user deforms and connects pieces to achieve the desired result, for example as in Fig. 1 for roads, or Fig. 10 for game levels. When modifications are made on a given piece, deformations and reshape are appropriately propagated along the chain of pieces. We can limit the propagation length during interactive manipulation; when the user releases the handles, the propagation is completed. We also cache the matrix factorizations to accelerate updates. Please see the video for example usage.

Mesh pieces and Interactive tool We have developed an interactive tool providing a simple, yet intuitive interface to assemble and reshape pieces. The interface allows the user to connect pieces (by their portals) and reshape them. To connect two pieces, the user simply clicks on a piece portal, which brings up a list of possible pieces with matching portals that can be connected to the selected piece (left hand side of the interface - see Fig. 1). Reshaping is possible by clicking on any vertex: when one vertex is selected as a handle, an axis-widget is shown, allowing the user to move the vertex. Changes made to this vertex propagate through all pieces affected. Besides moving vertices, the user can also add constraints between edges (rigidity values can be added interactively in a pop-up menu - Fig. 1).

Applications: House/Road Building and Doom Levels We first show an example of house building in Fig. 9. We then show an example with road blocks in Fig. 1. As we can see, quite complex road structures, with bridges, many-lane highways, overpasses etc. can be built from a small number of initial pieces. Fig. 10, shows game level pieces and the construction of a new modified level using our approach.

In these examples, mesh pieces had between 200-1400 vertices (450-4000 equations), creating the matrix takes from 40-800ms (unoptimized), and factorization takes between 20ms and 2.5s (for the bridge piece). For textures factorization takes 20-330ms with a maximum of almost 5000 equations.

6. Discussion and Conclusion

We have already discussed limitations specific to geometry reshape in Sec. 3.5. In terms of other limitations, we currently assume that the models are well formed in textured indexed face sets. We have found this to be a reasonable assumption for example with the game levels of "Doom" we tested here. For the custom-built pieces we also show here, no particular modifications were required in the modeler's workflow to produce models we could use. Nonetheless, it may be desirable to re-use model pieces which have

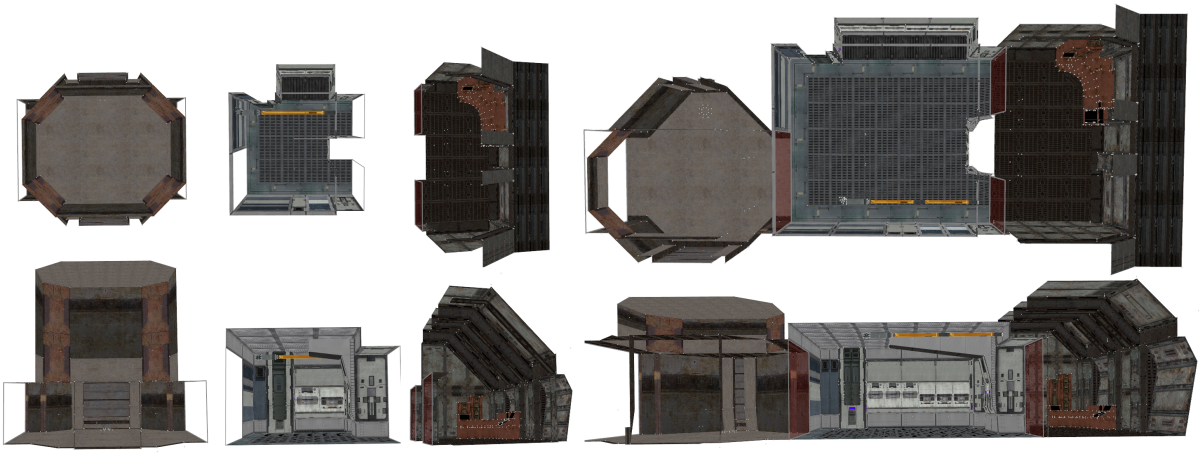

Figure 10: *Examples of our Doom level construction. We show the level building blocks and two views of a final construction made in tens of seconds. Note that these pieces do not trivially connect by construction. Textures and models from the game Doom III™. ©Id Software, all rights reserved.*

some inconsistencies in terms of connectivity information, for instance data from 3D scanners. We did some initial experimentation with such pieces, and we found that a simple voxelization approach to determine corners was sufficient to extract corners and connectivity. Evidently, a general robust solution is a very difficult problem with strong links to mesh repair techniques [Ju04]. Since our method is primarily designed to be applied to models created by 3D modeling software we do not expect this to be be a major limitation.

Another limitation is our assumption that each connected component of texture detail contains one tile. Evidently this may not be the case; A multi-label segmentation should be performed to identify this and extract the appropriate tiles.

Finally, we currently assume that portals of pieces are "compatible". Correctly treating incompatible portals is a hard problem; a more general "geometry matching" approach needs to be developed. While methods to treat general meshes exist (e.g., [FKS*04]), they are not necessarily adapted to architectural pieces. We are actively pursuing this direction of research.

Adding feedback between texture rigidity constraints and geometric reshape should be relatively straightforward to add. This would be particularly helpful in cases when we cannot identify detail tiles for examples.

To conclude, we have introduced a new approach which simultaneously treats geometry and texture reshape of architectural or structured man-made models. Our new algorithm enables modeling by adapting and varying existing pieces, and chaining them together, achieving rapid construction of complex models.

7. Acknowledgments

We would like to thank Fernanda Andrade Cabral for her help with modeling and texturing. We also thank Frédo Durand, Eugene Fiume and Olga Sorkine for their input, as well as the reviewers for their helpful comments and for suggesting many improvements to the paper.

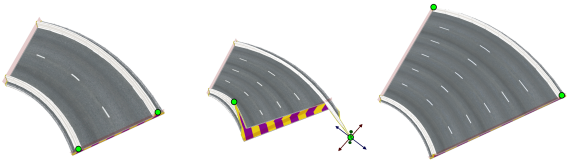

Figure 11: Left and middle: *The choice of handles results in undesired motions.* Right: *Thanks to interactive feedback the user easily chooses a better set of handles and achieves the desired reshape.*

References

[AS07] AVIDAN S., SHAMIR A.: Seam carving for content-aware image resizing. *ACM Trans. on Graphics (Proc. of SIGGRAPH 2007) 26*, 3 (2007).

[BBK05] BOTSCH M., BOMMES D., KOBBELT L.: Efficient linear system solvers for mesh processing. In *IMA Conference on the Mathematics of Surfaces* (2005), pp. 62–83.

[BFH05] BERNDT R., FELLNER D. W., HAVEMANN S.: Generative 3d models: a key to more information within less bandwidth at higher quality. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology* (2005), pp. 111–121.

[BJ01] BOYKOV Y. Y., JOLLY M. P.: Interactive graph cuts for optimal boundary region segmentation of objects in n-d images. vol. 1, pp. 105–112 vol.1.

[BMSX97] BORNING A., MARRIOTT K., STUCKEY P., XIAO Y.: Solving linear arithmetic constraints for user interface applications. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1997), ACM, pp. 87–96.

[BS08] BOTSCH M., SORKINE O.: On linear variational surface deformation methods. *IEEE Trans. on Vis. and Comp. Graphics 14*, 1 (2008), 213–230.

[DCOY03] DRORI I., COHEN-OR D., YESHURUN H.: Fragment-based image completion. *ACM Trans. on Graphics (Proc. of SIGGRAPH 2003) 22*, 3 (2003), 303–312.

[DMR97] DANIELS K., MILENKOVIC V., ROTH D.: Finding the largest area axis-parallel rectangle in a polygon. *Computational Geometry: Theory and Applications 7*, 1-2 (1997), 125–148.

[EA08] EA: Spore, 2008. http://www.spore.com.

[EL99] EFROS A. A., LEUNG T. K.: Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision* (September 1999), pp. 1033–1038.

[FKS*04] FUNKHOUSER T., KAZHDAN M., SHILANE P., MIN P., KIEFER W., TAL A., RUSINKIEWICZ S., DOBKIN D.: Modeling by example. *ACM Trans. on Graphics (Proc. of SIGGRAPH 2004)* (2004).

[Gle92] GLEICHER M. L.: Integrating constraints and direct manipulation. In *Proc. of the ACM SIGGRAPH Symp. on Interactive 3D Graphics* (New York, NY, USA, 1992), ACM, pp. 171–174.

[Ju04] JU T.: Robust repair of polygonal models. *ACM Trans. on Graphics 23*, 3 (2004), 888–895.

[KJS07] KRAEVOY V., JULIUS D., SHEFFER A.: Model composition from interchangeable components. In *Pacific Graphics* (2007).

[KSCOS08] KRAEVOY V., SHEFFER A., COHEN-OR D., SHAMIR A.: Non-homogeneous resizing of complex models. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH ASIA 2008) 27*, 5 (2008).

[Loa85] LOAN C. V.: On the method of weighting for equality-constrained least-squares problems. *SIAM Journal on Numerical Analysis 22*, 5 (1985), 851–864.

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. *ACM Trans. on Graphics (Proc. of SIGGRAPH 2008) 27*, 3 (Aug. 2008).

[Mer07] MERRELL P.: Example-based model synthesis. In *Proc. of the ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games* (2007).

[MM08] MERRELL P., MANOCHA D.: Continuous model synthesis. In *SIG - Asia* (New York, NY, USA, 2008), no. to appear, ACM SIGGRAPH, p. to appear.

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural modeling of buildings. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH 2006)* (2006), pp. 614–623.

[MZWG07] MÜLLER P., ZENG G., WONKA P., GOOL L. V.: Image-based procedural modeling of facades. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH 2007) 26*, 3 (2007), 85.

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., 1990.

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proc. SIGGRAPH '01* (2001), ACM, pp. 301–308.

[SCOL*04] SORKINE O., COHEN-OR D., LIPMAN Y., ALEXA M., RÖSSL C., SEIDEL H.-P.: Laplacian surface editing. In *SGP '04: Proc. EG/SIGGRAPH Symp. on Geometry processing* (2004), pp. 175–184.

[TBTS08] TAI Y.-W., BROWN M. S., TANG C.-K., SHUM H.-Y.: Texture amendment: Reducing texture distortion in constrained parameterization. *SIG - Asia to appear*, to appear (2008), to appear.

[TCR03] TOLEDO S., CHEN D., ROTKIN V.: Taucs: A library of sparse linear solvers, 2003. http://www.tau.ac.il/~stoledo/taucs/.

[TM98] TOMASI C., MANDUCHI R.: Bilateral filtering for gray and color images. In *ICCV '98: Proc. of the Int. Conf. on Comp. Vision* (1998), p. 839.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH'01* (2000), ACM SIGGRAPH, pp. 479–488.

[WTSL08] WANG Y.-S., TAI C.-L., SORKINE O., LEE T.-Y.: Optimized scale-and-stretch for image resizing. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH ASIA 2008) 27*, 5 (2008).

[XSF02] XU K., STEWART J., FIUME E.: Constraint-based automatic placement for scene composition. In *Graphics Interface* (2002), pp. 25–34.

[YZX*04] YU Y., ZHOU K., XU D., SHI X., BAO H., GUO B., SHUM H.-Y.: Mesh editing with poisson-based gradient field manipulation. In *Proc. SIGGRAPH* (2004), pp. 644–651.

[ZHW*06] ZHOU K., HUANG X., WANG X., TONG Y., DESBRUN M., GUO B., SHUM H.-Y.: Mesh quilting for geometric texture synthesis. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH 2006)* (2006), pp. 690–697.